The VolumePro Real-Time Ray-Casting System

The VolumePro Real-Time Ray-Casting System

Hanspeter Pfister * Jan Hardenbergh † Jim Knittel† Hugh Lauer† Larry Seiler†

Mitsubishi Electric

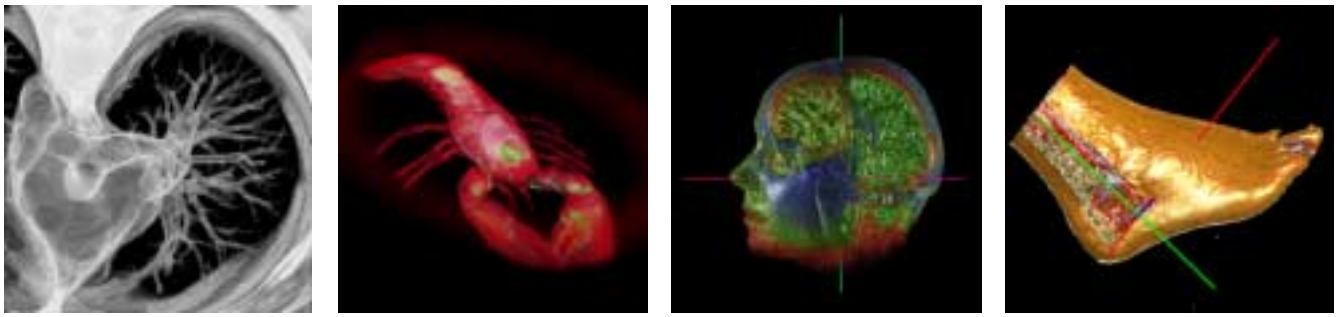


Figure 1: *Several volumes rendered on the VolumePro hardware at 30 frames per second.*

Abstract

This paper describes VolumePro, the world's first single-chip real-time volume rendering system for consumer PCs. VolumePro implements ray-casting with parallel slice-by-slice processing. Our discussion of the architecture focuses mainly on the rendering pipeline and the memory organization. VolumePro has hardware for gradient estimation, classification, and per-sample Phong illumination. The system does not perform any pre-processing and makes parameter adjustments and changes to the volume data immediately visible. We describe several advanced features of VolumePro, such as gradient magnitude modulation of opacity and illumination, supersampling, cropping and cut planes. The system renders 500 million interpolated, Phong illuminated, composited samples per second. This is sufficient to render volumes with up to 16 million voxels (e.g., 256^3) at 30 frames per second.

CR Categories: B.4.2 [Hardware]: Input/Output and Data Communications—Input/Output DevicesImage display; C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems—Real-time and embedded systems; I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processor;

Keywords: Graphics Hardware, Hardware Systems, Rendering Hardware, Rendering Systems, Volume Rendering

*MERL, 201 Broadway, Cambridge, MA 02139, pfister@merl.com

†Real Time Visualization, Mitsubishi Electric, 300 Baker Avenue, Concord, MA 01742, [jch,knittel,lauer,seiler]@merl.com

1 Introduction

Over the last decade, direct volume rendering has become an invaluable visualization technique for a wide variety of applications. Examples include visualization of 3D sampled medical data (CT, MRI), seismic data from oil and gas exploration, or computed finite element models. While volume rendering is very popular, the lack of interactive frame rates has limited its widespread use. To render one frame typically takes several seconds due to the tremendous storage and processing requirements. The three most promising approaches to achieve real-time frame rates for volume rendering are highly optimized software methods, use of 3D texture mapping hardware, or special-purpose volume rendering hardware.

Optimized software techniques for volume rendering require pre-computation and additional data storage, and they often sacrifice image quality for speed. Shear-warp rendering [11] is currently the fastest software algorithm. It achieves 1.1 Hz on a single 150MHz R4400 processor for a $256 \times 256 \times 225$ volume with 65 seconds of pre-processing time [10]. Unfortunately, pre-processing prohibits immediate visual feedback during parameter changes. It also prevents real-time visualization of dynamically changing volume data.

3D texture mapping hardware can be used for volume rendering using a method called planar texture resampling [4]. A volume is stored as a 3D texture and resampled during rendering by extracting textured planes parallel to the image plane. Interactive rendering rates have been achieved on the SGI Reality Engine [1, 2]. However, texture hardware does not support estimation of gradients that are required to identify surfaces for shading. Efforts to overcome these difficulties either require multi-pass rendering, are limited to iso-surfaces and do not extend to specular illumination [18], or require significant pre-processing [6]. Other restrictions, especially on PC graphics cards, include limited arithmetic precision for blending and interpolation and limited texture memory.

Special purpose hardware for volume rendering has been proposed by various researchers, but only a few machines have been implemented. VIRIM was built at the University of Mannheim, Germany [7]. The hardware consists of four VME boards and implements ray-casting. VIRIM achieves 2.5 frames/sec for 256^3 volumes. The VIZARD system of the University of Tübingen, Germany, implements perspective ray-casting and consists of two PCI

accelerator cards [9]. An FPGA-based system achieves up to 10 frames/sec for 256^3 volumes. The system, however, uses lossy data compression, and any changes in classification or shading parameters require lengthy pre-processing.

To overcome these limitations, we have developed VolumePro. VolumePro is the first single-chip real-time volume rendering system for consumer PCs. It does not require any pre-processing and performs a brute-force resampling of all voxels for each frame. This makes changes to the volume data immediately visible and allows the integration of VolumePro hardware into simulation systems (e.g., surgical simulation) or real-time acquisition devices (e.g., 3D ultrasound). VolumePro has hardware for gradient estimation, per-sample Phong illumination, and classification, and all parameters can be adjusted in real-time.

A VolumePro system consists of the VolumePro PCI card, a companion 3D graphics card, and software. The VolumePro PCI card contains 128 MB of volume memory and the vg500 rendering chip. The Volume Library Interface (VLI) is a collection of C++ objects and provides the application programming interface to the VolumePro features. The first VolumePro board was operational in April 1999 (see Figure 2). Production shipments started in June 1999 at an initial price comparable to high-end PC graphics cards.

Figure 2: *The VolumePro PCI card.*

VolumePro is based on the Cube-4 volume rendering architecture developed at SUNY Stony Brook [14]. Cube-4 requires, however, a large number of rendering and memory chips, many pins for inter-chip communication, and large on-chip storage for intermediate results. Consequently, we developed Enhanced Memory Cube-4 (EM-Cube) [13] to implement Cube-4 at lower cost.

VolumePro is the commercial implementation of EM-Cube, and it makes several important enhancements to its architecture and design. By giving up scalability we are able to fit four rendering pipelines on one chip (see Section 5). An on-chip voxel distribution network greatly simplifies communication between rendering pipelines (see Section 5.3). A novel block-and-bank skewing scheme takes advantage of the internal organization of modern SDRAM devices (see Section 5.2). VolumePro also implements several novel features, such as gradient magnitude modulation, supersampling, supervolumes, slicing, and cropping (see Section 4).

The reality of a tight production schedule, however, forced us to make compromises. VolumePro supports only 8- and 12-bit scalar voxels and no other voxel formats. It performs orthographic projections of rectilinear volume data sets. Perspective projections and intermixing of polygons and volumes were deemed too complex and were postponed for a future release of the system. This paper is the first detailed description of the VolumePro architecture and system. All images in this paper were rendered on a pre-production version of the VolumePro hardware at 30 frames/sec.

2 Rendering Algorithm

VolumePro implements ray-casting [12], one of the most commonly used volume rendering algorithms. Ray-casting offers high image quality and is easy to parallelize. The current version of VolumePro supports parallel projections of isotropic and anisotropic rectilinear volumes with scalar voxels.

To achieve uniform data access we use a ray-casting technique with hybrid object/image-order data traversal based on the shear-warp factorization of the viewing matrix [19, 15, 11] (see Figure 3). The volume data is defined in object coordinates (u, v, w), which are first transformed to isotropic object coordinates by the scale and shear matrix L. This allows to automatically handle anisotropic data sets, in which the spacing between voxels differs in the three dimensions, and gantry tilted data sets, in which the slices are sheared, by adjusting the warp matrix. We discuss gradient estimation in anisotropic and sheared volumes in Section 3.2.

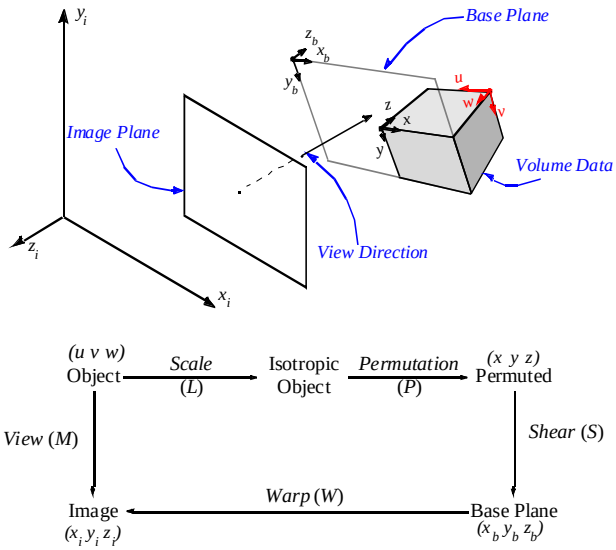


Figure 3: *Shear-warp factorization of the viewing matrix.*

The permutation matrix P transforms the isotropic object to permuted coordinates (x, y, z). The origin of permuted coordinates is the vertex of the volume nearest to the image plane and the z axis is the edge of the volume most parallel to the view direction.

The shear matrix S represents the rendering operation that projects points in the permuted volume space onto points on the base plane. In VolumePro this projection is performed using ray-casting. Instead of casting rays from image space (x_i, y_i, z_i), rays are sent into the data set from the base plane (x_b, y_b, z_b), which is the face of the volume data that is most parallel to the viewing plane. This approach guarantees that there is a one-to-one mapping of sample points to voxels [19, 15].

In contrast to the shear-warp implementation by Lacroute and Levoy [11], VolumePro performs tri-linear interpolation and allows rays to start at sub-pixel locations. This prevents view-dependent artifacts when switching base planes and accommodates supersampling of the volume data (see Section 4.1).

The base plane image is transformed to the image plane using the warp matrix $W = M \times L^{-1} \times P^{-1} \times S^{-1}$. VolumePro uses 2D texture mapping with bi-linear interpolation on a companion graphics card for this image warp (see Section 6). The additional 2D image resampling results in a slight degradation of image quality. It enables, however, an easy mapping to an arbitrary user-specified image size.

The main advantage of the shear-warp factorization is that voxels can be read and processed in planes of voxels, called slices, that are parallel to the base plane. Slices are processed in positive z direction. Within a slice, scanline of voxels (called voxel beams) are read from memory in top to bottom order. This leads to regular, object-order data access. In addition, it allows parallelism by having multiple rendering pipelines work on several voxels in a beam at the same time. This concept is explained further in Section 5. The next section describes a single VolumePro rendering pipeline.

3 The Ray-Casting Pipeline

A key characteristic of VolumePro is that each voxel is read from memory exactly once per frame. Therefore, voxel values must be recirculated through the processing stages of the VolumePro pipeline so that they become available for calculations precisely when needed.

A second characteristic of VolumePro is that the reading and processing of voxel data and calculation of pixel values on rays are highly pipelined. One VolumePro processing pipeline can accept a new voxel every cycle and can forward intermediate results to subsequent pipeline stages every cycle. The net effect is that VolumePro can render a volume data set at the speed of reading voxels.

Figure 4 illustrates a flow diagram of an idealized version of the VolumePro processing pipeline. In this figure, the flow and processing of voxel data is shown in black, and control information is shown in red. The figure shows interpolation followed by gradient estimation. In actual practice, the x and y gradients are computed following interpolation, but the z gradients are estimated before interpolation (see Section 3.2).

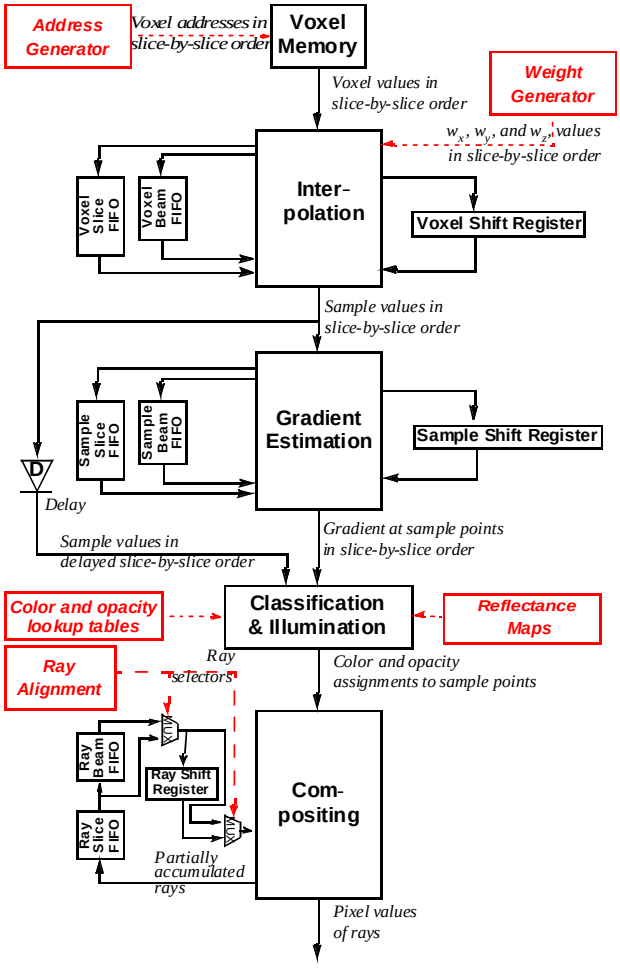


Figure 4: *Conceptual ray-casting pipeline.*

3.1 Interpolating Voxel Values

At the top of Figure 4 is Volume Memory, also called Voxel Memory. The current generation of VolumePro supports 8- and 12-bit scalar voxels. All internal voxel datapaths are 12-bits wide. Voxels are read from the memory and are presented to the Interpolation unit in the slice-by-slice, beam-by-beam order described in Section 2, one voxel per cycle. The Interpolation unit converts the stream of voxel values into a stream of sample values, also in slice-by-slice, beam-by-beam order, at the rate of one sample value per cycle. The interpolated samples are rounded to 12-bit values.

Each sample value is derived from its eight nearest neighboring voxels by tri-linear interpolation. Therefore, the Interpolation Unit must have on hand not only the current voxel from the input stream, but also all other voxels from the same tri-linear neighborhood. It does this by storing input voxels and other intermediate values in the Voxel Slice FIFO, Voxel Beam FIFO, and Voxel Shift Register. Each of these is a data storage element sized so that when a newly calculated value is inserted, it emerges again at precisely the time needed for a subsequent calculation.

Trilinear interpolation also requires a set of weights. These are generated by the Weight Generator in Figure 4. This generator is based on a digital differential analyzer (DDA) algorithm to calculate each new set of weights from the previous weights. Since all rays are parallel to each other and have exactly the same spacing as the pixel positions on the base plane (and hence, as the voxels of each slice), a single set of weights is sufficient for all of the samples of a slice.

3.2 Gradient Estimation

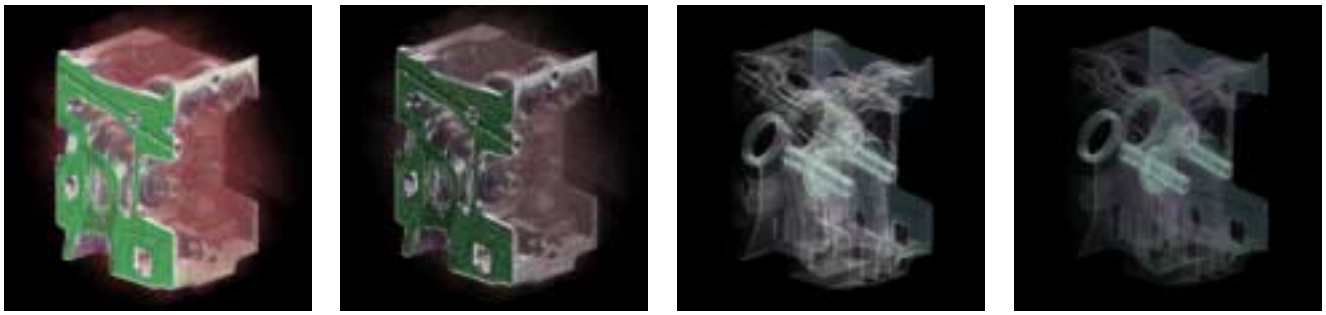
The output of the Interpolation unit is a stream of sample values, one per cycle in slice-by-slice, beam-by-beam order. This stream is presented to the Gradient Estimation unit for estimating the x, y and z gradients on each sample using central differences.

VolumePro computes gradients in two equivalent ways. In z direction, we interpolate differences between voxels to sample positions, and in x and y direction we take differences of interpolated samples. Both orders of computation yield equivalent results because difference and tri-linear interpolation are commutative linear operators.

The gradient at a particular sample point depends not only on data in the current slice but also on data in the previous and next slice. The next slice, of course, has not been read when the current sample value arrives at the input. Therefore, the Gradient Estimation unit operates one voxel, one beam, and one slice behind the Interpolation unit. That is, when sample $S_{(x+1),(y+1),(z+1)}$ arrives, the Gradient Estimation Unit finally has enough information to complete the estimation of the gradient at sample point $S_{x,y,z}$.

It does this by maintaining two full slices of buffering, plus two full beams of buffering, plus two extra samples. These are stored in the Sample Slice FIFOs, Sample Beam FIFOs, and Sample Shift Registers, respectively, which serve roughly the same functions as the Voxel Slice FIFO, Voxel Beam FIFO, and Voxel Shift Register. Each central difference gradient has signed 8-bit precision, meaning it can represent numbers in the range $[-127 \cdots +127]$.

VolumePro handles anisotropic and sheared volumes by treating them as isotropic and subsequently adjusting the warp matrix (see Section 2). The different voxel spacing and alignment leads to incorrect gradients, however, because the vector components are not



(a) None (b) Illumination (c) Opacity (d) Opacity and illumination

Figure 5: *Images rendered with gradient magnitude modulation of opacity or illumination. Notice how the appearance of surfaces changes.*

orthonormal in object space. We correct for these differences in software by adjusting the light positions every frame to compute the accurate dot product between gradient and light vectors for ambient and diffuse illumination. To correct the specular highlights we adjust the eye vector, used in the reflection vector hardware, every frame (see Section 3.5).

3.3 Gradient Magnitude

The gradient magnitude can be used to render multiple semi-transparent surfaces in volumes [12]. In VolumePro the sample opacity and illumination are optionally multiplied with the gradient magnitude. This modulation can be used to emphasize surface boundaries, to reduce the illumination of samples with small gradients, or to minimize the visual impact of noisy data.

Figure 5 shows four renderings of a CT scan of an engine block (256 x 256 x 110). Figures 5(a) and (b) show the difference between no modulation and gradient magnitude modulation of specular illumination. Regions of small gradient magnitude, such as noise, are de-emphasized in Figure 5(b). Figure 5(c) shows gradient magnitude modulation of opacity. Homogeneous regions with small gradient magnitude, such as the walls, are more transparent compared to Figure 5(a). Figure 5(d) shows the combined effect of modulating opacity and illumination by the gradient magnitude. Homogeneous regions are more transparent and the overall illumination is attenuated.

VolumePro derives a sample's gradient magnitude in two phases. First, the square of the gradient magnitude is computed by taking the sum of the squares of the gradient components. Then a Newton-Raphson iteration is used to compute the square-root of this value, resulting in an approximation of the gradient magnitude. The gradient components are in the range of $[-127 \cdots +127]$, making the maximum gradient magnitude value $\sqrt{3 \cdot 127^2}$. It is stored as an unsigned 8-bit number in the range of $[0 \cdots 220]$.

The gradient magnitude is then mapped to the range $[0 \cdots 255]$ by a lookup table (GmLUT). The table stores a user-specified piecewise linear function that can be used to highlight particular gradient magnitude values or to attenuate the modulation effect. The table is also used to automatically correct the gradient magnitudes in anisotropic volumes.

3.4 Assigning Color and Opacity

VolumePro assigns RGBA values to interpolated samples as opposed to classifying voxel values first and then interpolating RGBA. This produces the greatest accuracy for nonlinear mappings from data to color and opacity. It also reduces the number of interpolators by a factor of four. Unfortunately, it prevents direct rendering of volumes that have been pre-classified by a software segmentation algorithm into RGBA or material volumes [5].

The classification of sample values to RGBA in VolumePro can summarized as:

$$\alpha = \begin{cases} \mathrm{GmLUT}(|\mathrm{Gradient}|) \cdot \alpha\mathrm{LUT}(\mathrm{Sample}) & \text{if } G_o = 1 \\ \alpha\mathrm{LUT}(\mathrm{Sample}) & \text{if } G_o = 0 \end{cases}$$

$$\mathrm{SampleRGB} = \mathrm{colorLUT}(\mathrm{Sample}).$$

The actual classification of samples is a straightforward table lookup of the 12-bit sample value into a 4096×36 bit classification lookup table (LUT). The lookup table stores 36-bit RGBA values. Color values are stored with 8-bit precision for each of R, G, and B. α is stored with 12-bits precision for maximum accuracy during ray-casting of low opacity volumes.

The RGBA table is pre-computed and loaded into VolumePro prior to rendering. Color and opacity can be loaded separately. An additional on-chip RGBA table hides the latency of loading the tables through double buffering. Loading a new opacity table with $4k \times 12$ bit entries requires 2k 32-bit PCI transfers, or 246 KB/sec for 30 loads/sec. Updating the table every frame corrects the opacities for non-unit sample spacing during view changes, anisotropic volumes, or supersampling [5]. As mentioned in Section 3.3, the value of α may be optionally multiplied by the gradient magnitude, depending on the value of G_o.

3.5 Sample Illumination

VolumePro implements Phong illumination at each sample point at the rate of one illuminated sample per clock cycle. The shading calculation can be summarized as:

$$\begin{aligned} \mathrm{RGB} &= ((k_e + (I_d \cdot G_d \cdot k_d)) \cdot \mathrm{SampleRGB}) + \\ &\quad (I_s \cdot G_s \cdot k_s \cdot \mathrm{SpecularColor}), \end{aligned}$$

where:

$$\begin{aligned} I_d &= \mathrm{DiffuseReflectanceMap}(\mathrm{Gradient}), \\ I_s &= \mathrm{SpecularReflectanceMap}(\mathrm{ReflectionVector}), \\ G_{[d,s]} &= \begin{cases} \mathrm{GmLUT}(|\mathrm{Gradient}|) & \text{if } \mathrm{Gmim}_{[d,s]} = 1 \\ 1 & \text{if } \mathrm{Gmim}_{[d,s]} = 0 \end{cases} \end{aligned}$$

and where k_e, k_d, k_s, and specularColor are registers of VolumePro. The values of Gmim_d and Gmim_s enable or disable Gradient Magnitude Illumination Modulation for diffuse or specular illumination, respectively.

The sample color from the classification LUT is multiplied by k_e to produce the emissive color of the object. The diffuse contribution is obtained by multiplying a user-defined diffuse coefficient k_d with

the diffuse illumination value I_d and the sample color. The specular contribution is obtained by multiplying a user-defined specular color with the product of a specular coefficient k_s and the specular illumination value I_s.

The diffuse and specular illumination values I_d and I_s are looked up in reflectance maps, respectively. Each reflectance map is a precomputed table that stores the amount of illumination due to the sum of all of the light sources of the scene. The reflection map implementation supports an unlimited number of directional light sources, but no positional lights. Trading computation for table lookup leads to simpler logic than an arithmetic implementation of shading. The VLI software loads the diffuse and specular reflectance maps into VolumePro prior to rendering a frame.

Reflectance values are mapped onto six sides of a cube, indexed by the unnormalized gradient or reflection vectors [17]. The reflection vector for each sample is computed in hardware from the gradient and eye vectors. Using bi-linear interpolation among reflectance map values keeps the table size small without incurring noticeable visual artifacts [16]. The specular and diffuse map are implemented with 384 32-bit entries each. To load both tables takes 92 KB/sec for 30 loads/sec.

The reflectance maps need to be reloaded when the object and light positions change with respect to each other, or to correct the eye vector in anisotropic volumes (see Section 3.2). Because the reflection vector is computed in hardware, however, the reflectance maps do not have to be reloaded when the eye changes in isotropic volumes.

3.6 Accumulating Color Values along Rays

The output of the Classification and Shading unit is a stream of color and opacity values at sample points in slice-by-slice, beam-by-beam order. This stream is fed into the Compositing unit for accumulation into the pixel values of the rays. Samples along rays arrive in front-to-back order. The compositing unit that works on a particular ray also gets samples from other rays, due to the slice-order traversal. Thus, the samples must be buffered until the color and opacity values of the next sample point along the ray arrive in the stream.

The Ray Slice FIFO, Ray Beam FIFO, and Ray Shift Register hold the values of the partially accumulated rays for this purpose. Although these look somewhat like the Voxel Slice FIFO, Voxel Beam FIFO, and Voxel Shift Register, their functions are different because of the cyclical nature of compositing. To understand this, observe that the compositing operation for sample $S_{x,y,z}$ requires as input the result of compositing one of $S_{x,y,(z-1)}$, $S_{(x-1),y,(z-1)}$, $S_{x,(y-1),(z-1)}$, or $S_{(x-1),(y-1),(z-1)}$, depending on the view direction and the value of z. That is, the predecessor value required as input to a particular compositing operation is the result of one of four compositing operations of the previous slice of samples. The selection of which particular one falls under the control of the Ray Alignment unit near the bottom of Figure 4, which drives two multiplexers (labeled MUX in the figure).

The Ray Slice FIFO stores partially accumulated pixel values of rays and makes them available for compositing with color and opacity values from the next slice. The Ray Beam FIFO stores the same values for a further beam time, so that in case the rays angle downward, input values can be obtained from the beam above and before the current sample. Likewise, the Ray Shift Register stores the same value for one additional cycle, in case the input value needs to be obtained from a sample to the left of the current one.

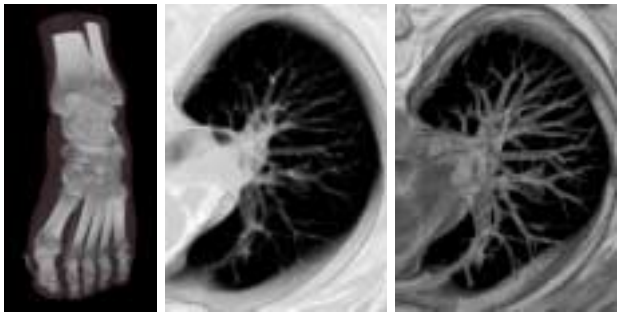
In addition to alpha blending, VolumePro supports Minimum and Maximum Intensity Projections (MIP) (see Table 1). In the table, $C_{\rm acc}$ and $\alpha_{\rm acc}$ are the accumulated color and opacity, respectively. VolumePro uses 12 bits of precision for $\alpha, C_{\rm acc}, \alpha_{\rm acc}$, and all intermediate compositing values for correct α-blending in low-opacity volumes and very large data sets.

Blend Mode	Functions
Front-to-back α-blending	$C_{\rm acc} += (1 - \alpha_{\rm acc}) \times (\alpha_{\rm sample} C_{\rm sample})$ $\alpha_{\rm acc} += (1 - \alpha_{\rm acc}) \times \alpha_{\rm sample}$
Minimum Intensity	if (sampleValue $<$ minValue): $C_{\rm acc} = C_{\rm sample}$; minValue = sampleValue;
Maximum Intensity	if (sampleValue $>$ maxValue): $C_{\rm acc} = C_{\rm sample}$; maxValue = sampleValue;

Table 1: *Blending modes of VolumePro.*

Finally, after the color and opacity values of all of the sample points on an individual ray have been accumulated, the resulting pixel value of that ray is output. This may occur when a ray passes through the back of the volume – i.e., with a maximum value of z – or when it passes through a side face of the volume. Base plane pixel values are written to Pixel Memory and then transferred to a companion 3D graphics card for the final image warp. The image warp is performed by 2D texture mapping hardware on the companion 3D graphics card. The base plane image is defined as a texture of a polygon corresponding to the base plane bounding box. The 3D graphics card transforms and bi-linearly resamples this textured polygon to the final image position.

Figure 6(a) shows a foot (152 x 261 x 200) of the visible man CT data set rendered with Maximum Intensity Projection (MIP). Figure 6(b) shows the CT scan of a lung (256 x 256 x 115) with low-opacity alpha blending and no illumination. Figure 6(c) shows the same dataset, but with illumination and gradient magnitude modulation of opacity.



(a) (b) (c)

Figure 6: *(a) MIP. (b) Alpha blending, no illumination. (c) Alpha blending, illumination, gradient magnitude modulation of opacity.*

4 Advanced Features of VolumePro

This section describes several additional features that have some impact on the architecture of VolumePro. These include supersampling, supervolumes (volumes larger than 256 voxels in any dimension), subvolumes, cropping and cut planes. We are not aware of any previous implementation of these features in special-purpose volume rendering hardware.

4.1 Supersampling

Supersampling [8] improves the quality of the rendered image by sampling the volume data set at a higher frequency than the voxel

spacing. In the case of supersampling in the x and y directions, this would result in more samples per beam and more beams per slice, respectively. In the z direction, it results in more sample slices per volume.

VolumePro supports supersampling in hardware only in the z direction. Additional slices of samples are interpolated between existing slices of voxels. The software automatically corrects the opacity according to the viewing angle and sample spacing by reloading the opacity table (see Section 3.4).

Figure 7 shows the CT scan of a foot (152 x 261 x 200) rendered with no supersampling (left) and supersampling in z by 3 (right). The artifacts in the left image stem from the insufficient sampling rate to capture the high frequencies of the foot surface. Notice the reduced artifacts in the supersampled image. VolumePro supports up to eight times supersampling.

Figure 7: *No supersampling (left) and supersampling in z (right).*

The impact of supersampling on the VolumePro processing pipelines is minimal. Since it is necessary to have voxels from two slices to interpolate in the z direction, it is sufficient to do several of these interpolations from the same pair of slices before moving on to another slice. Since samples pass from the Interpolation unit in slice-by-slice order, the only impact is that the voxel memory stages of the pipeline must stall and wait for the additional, supersampled slices to clear. Thus, if we supersample by a factor of k, the rendering rate in frames per second is reduced by $\frac{1}{k}$.

With the current implementation of VolumePro, supersampling in the x and y directions can be implemented by repeatedly rendering a volume with slightly different ray offsets on the base plane in the x and y dimensions. The VolumePro hardware supports initial ray offsets at sub-pixel accuracy. The VolumePro software provides the necessary support to automatically render several frames and to blend them into a final supersampled image.

4.2 Supervolumes and Subvolumes

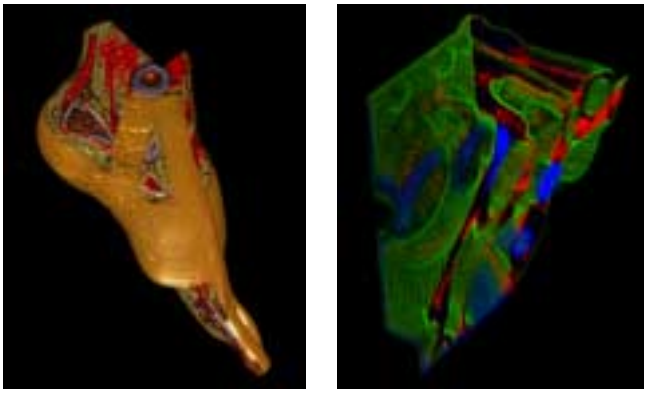
Volumes of arbitrary dimensions can be stored in voxel memory without padding. Because of limited on-chip buffers, however, the VolumePro hardware can only render volumes with a maximum of 256 voxels in each dimension in one pass. In order to render a larger volume (called a supervolume), software must first partition the volume into smaller blocks. Each block is rendered independently, and their resulting images are combined in software.

The VolumePro software automatically partitions supervolumes, takes care of the data duplication between blocks, and blends intermediate base planes into the final image. Blocks are automatically swapped to and from host memory if a supervolume does not fit into the 128 MB of volume memory on the VolumePro PCI card. There is no limit to the size of a supervolume, although, of course, rendering time increases due to the limited PCI download bandwidth.

Volumes with less than 256 voxels in each dimension are called subvolumes. VolumePro's memory controller allows reading and writing single voxels, slices, or any rectangular slab to and from Voxel Memory. Multiple subvolumes can be pre-loaded into volume memory. Subvolumes can be updated in-between frames. This allows dynamic and partial updates of volume data to achieve 4D animation effects. It also enables loading sections of a larger volume in pieces, allowing the user to effectively pan through a volume. Subvolumes increase rendering speed to the point where the frame rate is limited by the base plane pixel transfer and driver overhead, which is currently at 30 frames/sec.

4.3 Cropping and Cut Planes

VolumePro provides two features for clipping the volume data set called cropping and cut planes. These make it possible to visualize slices, cross-sections, or other portions of the volume, thus providing the user an opportunity to see inside in creative ways. Figure 8(a) shows an example of cropping on the CT foot of the visible man. Figure 8(b) shows a cut plane through the engine data.



(a) (b)

Figure 8: *(a) Cropping. (b) Cut plane.*

Cropping introduces multiple clipping planes parallel to the volume faces. Figure 9 shows a conceptual diagram and several examples, and Figure 8(a) shows an example image.

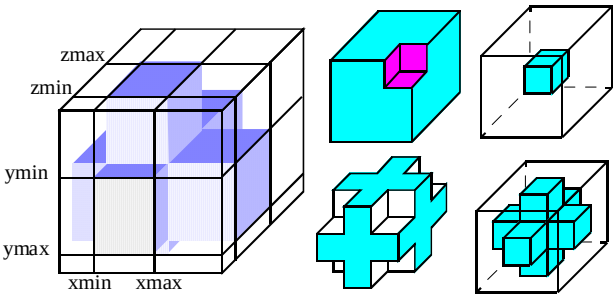


Figure 9: *Cropping of a volume data set and cropping examples.*

Three slabs, one parallel to each of the three axes of the volume data set, are defined by six registers $x_{\min}, x_{\max}, y_{\min}, y_{\max}, z_{\min}$, and $z_{\max}$. Slab S_x is the set of all points (x, y, z) such that $x_{\min} \leq x \leq x_{\max}$, slab S_y is the set of all points such that $y_{\min} \leq y \leq y_{\max}$, and slab S_z is the set of all points such that $z_{\min} \leq z \leq z_{\max}$. Slabs may be combined by taking intersections, unions, and inverses to define regions of visibility of the volume data set. A sample at position (x, y, z) is visible if and only if it falls in this region of visibility. The logic for cropping is implemented in the Compositing unit, which ignores invisible samples.

In Figure 9, the shaded part of the volume remains visible and the remainder is invisible. Alternatively, the same slabs could be combined in a different way so that only the intersection of the three

slabs is visible. Note that the cropping planes defining the slabs may fall at arbitrary voxel positions. A side effect is that cropping is an easy way of specifying a rectilinear region of interest (ROI).

The second form of clipping is the cut plane. VolumePro supports a single cut plane with arbitrary thickness and orientation. Samples are visible only if they lie between the two parallel surfaces of the cut plane; alternatively, samples are visible only if they are outside of the cut plane. Figure 10 illustrates a cut plane and its transition region.

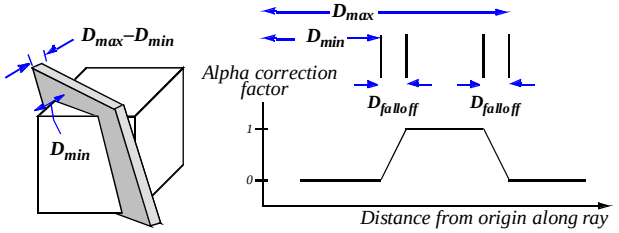


Figure 10: *Cut plane with transition regions.*

The two parallel faces of the cut plane are given by the plane equations:

$$\begin{aligned} Ax + By + Cz - D_{\min} &= 0 \\ Ax + By + Cz - D_{\max} &= 0, \end{aligned}$$

where $D_{\min} \leq D_{\max}$. That is, $D_{\min}$ measures the distance from the origin to one face of the cut plane, and $D_{\max}$ measures the distance from the origin to the other face. The thickness of the cut plane is $D_{\max} - D_{\min}$.

To allow for smooth cuts, a falloff parameter (D_{falloff}) specifies the width of the transition from full opacity to none (see Figure 10). Outside the cut plane, α is forced to zero. In the transition regions, α is multiplied by a correction factor. This correction factor increases linearly from zero to one. In the interior of the cut plane, α is simply the value from the shading stage. The α correction logic for the cut plane is in the Compositing unit.

As shown in Figure 1 on the right, VolumePro also has a 3D cursor feature that inserts a hardware generated, software controlled cursor into the volume data set being rendered. The 3D cursor allows users to explore and identify spatial relationships within the volume data. The samples for the cursor are generated in the Compositing unit and are blended into the volume data by α-blending.

5 vg500 Chip Architecture

The rendering engine of the VolumePro system is the vg500 chip with four parallel rendering pipelines. It is an application specific integrated circuit (ASIC) with approximately 3.2 million random logic transistors and 2 Mbits of on-chip SRAM. It is fabricated in 0.35 μ technology and runs at 125 MHz clock frequency. In this section we discuss a number of practical considerations that affect the architecture of the vg500 ASIC and that lead to modifications of the idealized pipeline of Section 3.

5.1 Parallel Pipelines

To render a data set of 256^3 voxels at 30 frames per second, VolumePro must be able to read $256^3 \times 30$ voxels per second – that is, approximately 503 million voxels per second. In the current implementation of VolumePro, each pipeline operates at 125 MHz and can accept a new voxel every cycle. Thus, achieving real-time rendering rates requires four pipelines operating in parallel. These can process $4 \times 125 \cdot 10^6$ or 500 million voxels per second.

The arrangement of the four pipelines in the vg500 chip architecture is illustrated in Figure 11. They work on adjacent voxel or

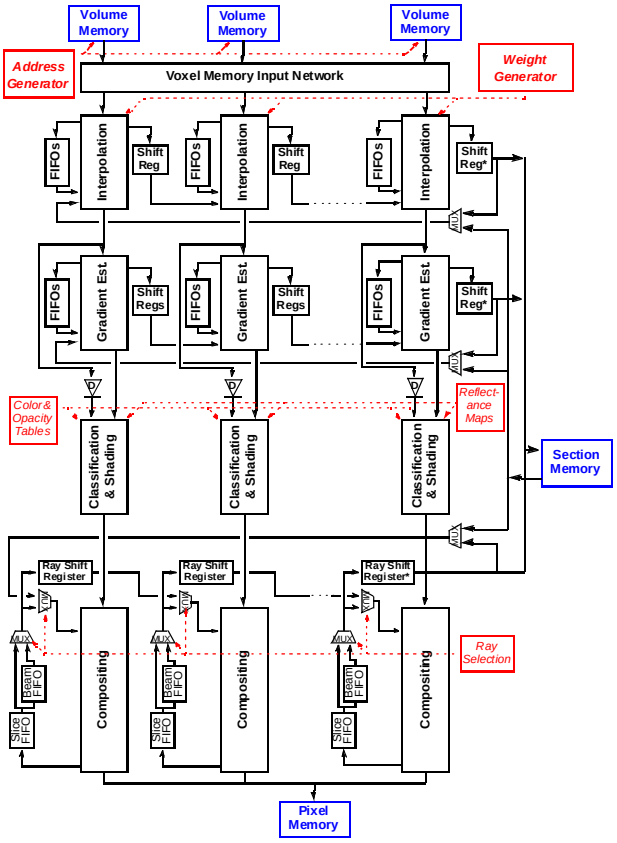


Figure 11: *The four parallel pipelines of the vg500 ASIC. Modules shown in blue are off-chip memory.*

sample values in the x direction. Each pipeline is connected to its neighbor by means of the shift registers in each of the major units. Whereas in Figure 4 the shift registers recirculate values that are needed in the next cycle in the x direction, in Figure 11 they provide values to their neighbors in the same cycle. The exceptions are the shift registers of the rightmost pipeline, each of which is marked with an asterisk. These act as in Figure 4 to recirculate values needed in the next cycle, but they send those values to the leftmost pipeline. An off-chip Voxel Memory supplies data to all of the pipelines, and they all write pixel values to an off-chip Pixel Memory. The additional datapaths on the right and the off-chip Section Memory will be discussed in Section 5.4.

5.2 Voxel Memory Organization

The vg500 chip has four 16-bit memory interfaces to Voxel Memory. A typical Voxel Memory configuration consists of four Synchronous Dynamic Random Access Memory (SDRAM) modules. The current implementation of VolumePro uses 64-megabit SDRAMs that provide burst mode access at 125 MHz. Voxel Memory can be extended to four SDRAMs per memory interface. Thus, sixteen SDRAMs provide 1024 megabits (i.e., 128 megabytes) of voxel storage. This is sufficient to hold a volume data set with 128 million 8-bit voxels or four 256^3 volumes with 12-bit voxels. Four 125 MHz memory interfaces can read Voxel Memory at a burst rate of 4×125 million (i.e., 500 million) voxels per second, provided that voxels can be organized appropriately in memory.

There are three architectural challenges. First, voxels have to be organized so that data is read from Voxel Memory in bursts of eight or more voxels with consecutive addresses. This is done by arranging voxels in miniblocks. A miniblock is a $2 \times 2 \times 2$ array of voxels stored linearly in a single memory module at consecutive memory addresses. All data is read in bursts of the size of a miniblock. This is much more efficient than the arbitrary memory accesses that resulted from voxel skewing in Cube-4 [14].

Second, miniblocks themselves have to be distributed across memory modules in such a way that groups of four adjacent miniblocks in any direction are always stored in separate memory modules, avoiding memory access conflicts. This is done by skewing [3]. Instead of skewing blocks, as in EM-Cube [13], this skewing technique is applied to miniblocks in VolumePro. It ensures that four adjacent miniblocks, i.e., miniblocks parallel to any axis, are always in different memory modules, no matter what the view direction is.

Third, miniblocks within a memory module must be further skewed so that adjacent miniblocks never fall into the same memory bank of an SDRAM chip. 64-megabit SDRAMs have four internal memory banks. $4 \times 4 \times 4$ cubes of miniblocks are skewed across the four memory banks, thus allowing back-to-back accesses to miniblocks in any traversal order with no pipeline delays.

A voxel with coordinates (u, v, w) has miniblock coordinates $(u_m, v_m, w_m) = (u/2, v/2, w/2)$, which are mapped to one of the memory modules and memory banks as follows:

$$\begin{aligned} \text{Module} &= (u_m + v_m + w_m) \bmod 4 \\ \text{Bank} &= ((u_m \div 4) + (v_m \div 4) + (w_m \div 4)) \bmod 4, \end{aligned}$$

where mod denotes the modulus operator and $\div$ the integer division operator. This is illustrated in Figure 12.

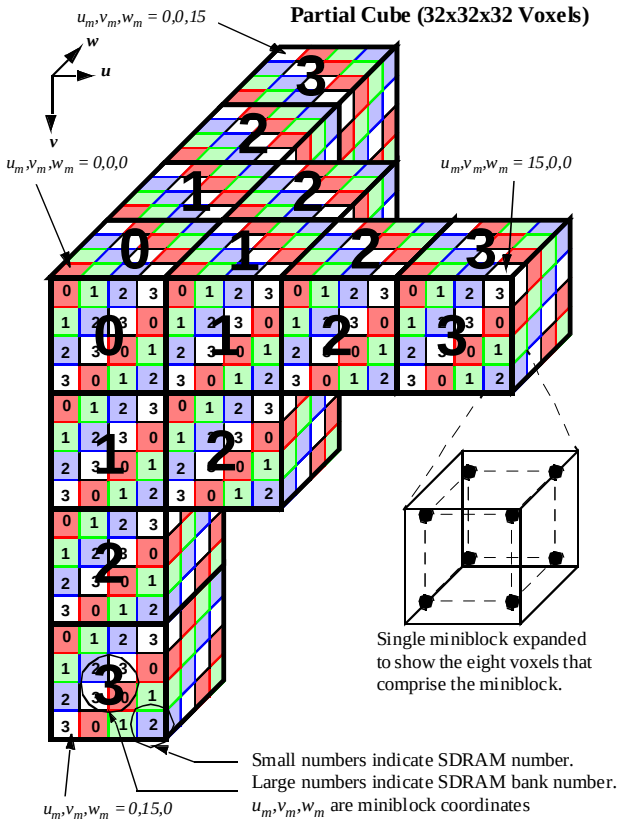


Figure 12: *Organization of miniblocks in Voxel Memory.*

In order to ensure that miniblocks are properly aligned for optimum SDRAM performance, the Voxel Memory must be allocated such that all dimensions of the volume object appear to be multiples of $2 \times 4 \times 4 = 32$ voxels. Arbitrary volumes are stored in Voxel Memory with dimensions that are multiples of 32 voxels; they are then automatically cropped to their original size during rendering. The vg500 ASIC contains all necessary hardware to arrange voxels into miniblocks and to skew and deskew miniblocks during data transfers to and from Voxel Memory.

5.3 Voxel Input To Rendering Pipelines

The description of the rendering pipeline in Section 3 assumes that the Interpolation and Gradient Estimation units read voxels in the slice-by-slice, beam-by-beam order described in Section 2. The Voxel Memory Input Network, shown in Figure 11, distributes data from the four Voxel Memory interfaces to the slice buffers. From there it can be read slice-by-slice and beam-by-beam by the four rendering pipelines.

Voxels in a miniblock are always read out of voxel memory in the same order because they are stored sequentially to take advantage of the burst capabilities of the SDRAMs. The miniblocks need to be re-arranged, based on the current view direction, so that the positions of the voxels correspond to the canonical positions assumed for tri-linear interpolation. This is achieved using simple miniblock reorder logic in each memory interface.

Because of the skewing of miniblocks in Voxel Memory, the four miniblocks output by the Voxel Memory controllers must be de-skewed and potentially re-shuffled based upon the chosen view direction. This is required so that the left-most voxel along the x axis flows down the leftmost pipeline and adjacent voxels in x flow down in neighboring pipelines. This is achieved with a global Voxel Memory Input Network that connects the four memory interfaces to the four rendering pipelines.

After the de-skewing network, miniblocks are written into the voxel slice buffers of the four rendering pipelines. Since data is read from voxel memory in miniblocks and processed by the pipelines as slices, a method exists to convert from one to another. The Voxel Memory interface actually reads two slices of voxels at a time because of the memory organization in miniblocks. Three slices of data must be stored so that the z gradients can be computed using central differences. After reading two slices of miniblocks and writing that data into four slice buffers, all necessary data now exists to compute interpolated samples and gradients.

5.4 Sectioning

To keep the amount of on-chip memory for the various FIFOs within reasonable limits, the volume data set is partitioned into sections and some intermediate values are off-loaded to external memory between the processing of sections. VolumePro implements sectioning only in the x direction. Each section is 32 voxels wide, corresponding to the memory organization outlined in Section 5.2. Instead of exchanging pixels between sections as in EM-Cube [13], VolumePro exchanges intermediate pipeline values between sections.

The impact of sectioning on the pipeline architecture is illustrated in Figure 11. In the right-hand pipeline of Figure 11 the shift registers are modified to optionally write values to Section Memory at the end of a section. The left-hand pipeline either reads values from Section Memory (at the beginning of a section) or accepts them from the right-hand pipeline. In effect, the Section Memory becomes a big, external FIFO capable of holding values that need to be recirculated in the x direction. Exactly the same values are written to, then read from, Section Memory as would have been passed

to the next pipeline. Consequently, no approximation is made and no visual artifacts are introduced at section boundaries.

The voxel traversal order is therefore modified. In particular, voxels are read in slice-by-slice order within a single section. Similarly, within slices, voxels are read in beam-by-beam order, but with beams spanning only the width of a section. The full section is processed front-to-back, and the intermediate values needed for rendering near the right boundary of the section are written to Section Memory. Then the next section is processed. Values are read from the Section Memory in the same order that they had been written, and they are passed through the multiplexers of Figure 11 to the left pipeline. As far as the rendering algorithm is concerned, it is as if the values had been generated on the previous cycle and passed directly. That is, the same values are calculated in either case, but in different orders.

6 VolumePro PCI Card

The VolumePro board is a PCI Short Card with a 32-bit 66 MHz PCI interface (see Figure 2). Production shipments started in June 1999 at an initial price comparable to high-end PC graphics cards. The board contains a single vg500 rendering ASIC, twenty 64 Mbit SDRAMs with 16-bit datapaths, clock generation logic, and a voltage converter to make it 3.3 volt or 5 volt compliant. Figure 13 shows a block diagram of the components on the board and the busses connecting them.

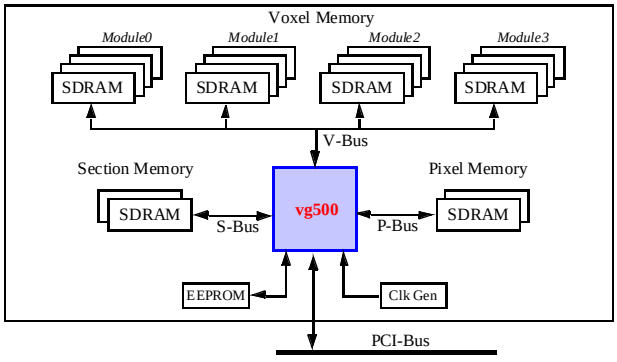


Figure 13: *VolumePro PCI board diagram.*

The vg500 ASIC interfaces directly to the system PCI-Bus. Access to the vg500's internal registers and to the off-chip memories is accomplished through the 32-bit 66 MHz PCI bus interface. The peak burst data rate of this interface is 264 MB/sec. Some of this bandwidth is consumed by image upload, some of it by other PCI system traffic. We currently estimate about 100 MB/sec available bandwidth for loading volume data from main memory.

Most registers are write-only and are memory-mapped via their own PCI base address register. Volume, pixel, and section memory are directly read/write memory-mapped into the PCI address space. The vg500 chip status is checked either by interrupts or by polling an on-chip status register. Alternatively, a copy of the status register gets DMA'ed to host memory when it changes. VolumePro supports many standard 16-bit and 32-bit pixel formats with on-the-fly conversion between formats during reads or writes.

The size of voxel memory is 128 MBytes, organized as four groups with four SDRAMs each. Two 64 Mbit SDRAMs make up Section Memory and two 64 Mbit SDRAMs of Pixel Memory contain the rendered base plane image. They can hold up to sixteen base planes, each with up to 512×512 32-bit RGBA pixels. This allows double-buffering of several base planes on the PCI card and pipelined retrieval, warping, and blending of images. The base plane pixels are transferred over the PCI bus to a companion 3D graphics card for the final image warp and display. The transfer of pixels from VolumePro, however, is the only integration with the rest of the graphics system. In particular, the current generation of the system does not perform any intermixing of polygons and volumes.

Several VolumePro PCI cards can be connected to a high-speed interconnect network for higher performance. Alternatively, several vg500 ASICs and their Voxel Memories can be integrated onto a single multi-processing rendering board. Volume data can be rendered in blocks, similar to supervolumes, on different boards or ASICs using coarse-grain parallelism. At the present time, however, we have no plans to implement such a multi-processing system.

7 VLI - The Volume Library Interface

Figure 14 shows the software infrastructure of the VolumePro system. The VLI API is a set of C++ classes that provide full access to

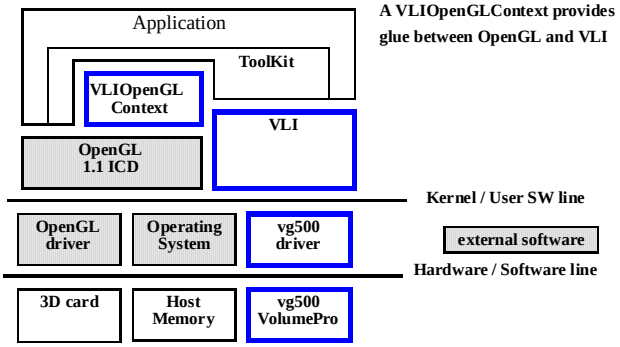


Figure 14: *Software infrastructure of the VolumePro system.*

the vg500 chip features. VLI does not replace an existing graphics API. Rather, VLI works cooperatively with a 3D graphics library, such as OpenGL, to manage the rendering of volumes and displaying the results in a 3D scene. We envision higher level toolkits and scene graphs on top of the VLI to be the primary interface layer to applications. The VLI classes can be grouped as follows:

- Volume data handling. VLIVolume manages voxel data storage, voxel data format, and transformations of the volume data such as shearing, scaling, and positioning in world space.
- Rendering elements. There are several VLI classes that provide access to the features described in Sections 3 and 4, such as color and opacity lookup tables, cameras, lights, cut planes, clipping, and more.
- Rendering context. The VLI class VLIContext is a container object for all attributes needed to render the volume. It is used to specify the volume data set and all rendering parameters (such as classification, illumination, and blending) for the current frame.

The VLI automatically computes reflectance maps based on light placement, sets up α-correction based on viewing angle and sample spacing, supports anisotropic and gantry-tilted data sets by correcting the viewing and image warp matrices, and manages supervolumes, supersampling, and partial updates of volume data. In addition, there are VLI functions that provide initialization, configuration, and termination for the VolumePro hardware.

8 Conclusions

This paper describes the algorithm, architecture, and features of VolumePro, the world's first single-chip real-time volume rendering system. The rendering capabilities of VolumePro – 500 million tri-linear, Phong illuminated, composited samples per second – sets a new standard for volume rendering on consumer PCs. Its core features, such as on-the-fly gradient estimation, per-sample Phong illumination with arbitrary number of light sources, 4K RGBA classification tables, α-blending with 12-bit precision, and gradient magnitude modulation, put it ahead of any other hardware solution for volume rendering. Additional features, such as supersampling, supervolumes, cropping and cut planes, enable the development of feature-rich, high-performance volume visualization applications.

Some important limitations of VolumePro are the restriction to rectilinear scalar volumes, the lack of perspective projections, and no support for intermixing of polygons and volume data. We believe that mixing of opaque polygons and volume data can be achieved by first rendering geometry, transferring z buffer values from the polygon card to the volume renderer, and then rendering the volume starting from these z values. Future versions of the system will support perspective projections and several voxel formats, including pre-classified material volumes and RGBA volumes. The limitation to rectilinear grids is more fundamental and hard to overcome.

We hope that the availability of VolumePro will spur more research in new and innovative interaction techniques for volume data, such as interactive experimentation with rendering parameters. This may lead to new solutions for difficult problems, such as data segmentation and transfer function design. Other areas for future research are hardware support for irregular grid rendering, accurate iso-surface rendering, and integration of polygon rasterization and texturing into volume rendering systems. We are currently working on a next generation system with more features while continually increasing the performance and reducing the cost.

9 Acknowledgments

We would like to thank the Volume Graphics Engineering team, especially Bill Peet and Beverly Schultz. Ingmar Bitter, Frank Dachille, Urs Kanus, Chris Kappler, and Dick Waters contributed to the ideas that are implemented in the system. Thanks also to Forrester Cole, voxel wrangler and image master. Thanks to the reviewers for their constructive comments, and to Vikram Simha, Ken Correl, and Jennifer Roderick for proofreading the paper. Special thanks to Arie Kaufman for having a vision 15 years ago and consistently making a dream come true.

References

[1] K. Akeley. RealityEngine graphics. In *Computer Graphics*, Proceedings of SIGGRAPH 93, pages 109–116, August 1993.

[2] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *1994 Workshop on Volume Visualization*, pages 91–98, Washington, DC, October 1994.

[3] D. Cohen and A. Kaufman. A 3D skewing and de-skewing scheme for conflict-free access to rays in volume rendering. *IEEE Transactions on Computers*, 44(5):707–710, May 1995.

[4] T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texture mapping hardware. Technical Report TR93-027, Department of Computer Science at the University of North Carolina, Chapel Hill, 1993.

[5] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics*, 22(4):65–74, August 1988.

[6] A. Van Gelder and K. Kim. Direct volume rendering with shading via three-dimensional textures. In *ACM/IEEE Symposium on Volume Visualization*, pages 23–30, San Francisco, CA, October 1996.

[7] T. Guenther, C. Poliwoda, C. Reinhard, J. Hesser, R. Maenner, H.-P. Meinzer, and H.-J. Baur. VIRIM: A massively parallel processor for real-time volume visualization in medicine. In *Proceedings of the 9th Eurographics Workshop on Graphics Hardware*, pages 103–108, Oslo, Norway, September 1994.

[8] P. Haeberli and K. Akeley. The accumulation buffer; hardware support for high-quality rendering. In *Computer Graphics*, volume 24 of *Proceedings of SIGGRAPH 90*, pages 309–318, Dallas, TX, August 1990.

[9] G. Knittel and W. Strasser. Vizard – visualization accelerator for real-time display. In *Proceedings of the Siggraph/Eurographics Workshop on Graphics Hardware*, pages 139–146, Los Angeles, CA, August 1997.

[10] P. Lacroute. Analysis of a parallel volume rendering system based on the shear-warp factorization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):218–231, September 1996.

[11] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transform. In *Computer Graphics*, Proceedings of SIGGRAPH 94, pages 451–457, July 1994.

[12] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8(5):29–37, May 1988.

[13] R. Osborne, H. Pfister, H. Lauer, N. McKenzie, S. Gibson, W. Hiatt, and T. Ohkami. EM-Cube: An architecture for low-cost real-time volume rendering. In *Proceedings of the Siggraph/Eurographics Workshop on Graphics Hardware*, pages 131–138, Los Angeles, CA, August 1997.

[14] H. Pfister and A. Kaufman. Cube-4 – A scalable architecture for real-time volume rendering. In *1996 ACM/IEEE Symposium on Volume Visualization*, pages 47–54, San Francisco, CA, October 1996.

[15] P. Schröder and G. Stoll. Data parallel volume rendering as line drawing. In *1992 Workshop on Volume Visualization*, pages 25–31, Boston, MA, October 1992.

[16] J. van Scheltinga, J. Smit, and M. Bosma. Design of an on-chip reflectance map. In *Proceedings of the 10th Eurographics Workshop on Graphics Hardware*, pages 51–55, Maastricht, The Netherlands, August 1995.

[17] D. Voorhies and J. Foran. Reflection vector shading hardware. In *Computer Graphics*, Proceedings of SIGGRAPH 94, pages 163–166, Orlando, FL, July 1994.

[18] R. Westerman and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics*, Proceedings of SIGGRAPH 98, pages 169–177, 1998.

[19] R. Yagel and A. Kaufman. Template-based volume viewing. *Computer Graphics Forum, Proceedings Eurographics*, 11(3):153–167, September 1992.